

# A LOCALITY OPTIMIZING ALGORITHM FOR DEVELOPING STREAM PROGRAMS IN IMAGINE

Jing Du, Xuejun Yang, Canqun Yang, Xiaobo Yan, Yu Deng

*School of Computer, National University of Defense Technology, Changsha 410073, China*

**Abstract:** In this paper, we explore a novel locality optimizing algorithm for developing stream programs in Imagine to sustain high computational ability. Our specific contributions include that we formulate the relationship between streams and kernels as a Data&Computation Matrix (D&C Matrix), and present the key techniques for locality enhancement based on this matrix. The experimental results on five representative scientific applications show that our algorithm can effectively improve the computational intensiveness and avoid the utilization of index streams to achieve high locality in LRF and SRF.

**Key words:** locality; Imagine; D&C Matrix; computational intensiveness; basic stream.

## 1. INTRODUCTION

Imagine is a programmable stream processor that implements an efficient memory hierarchy including several local register files (LRFs) with a 256-word scratchpad unit (SP), a 128 KB stream register file (SRF) and off-chip DRAM to sustain high computations<sup>1,2</sup>. The stream applications on Imagine are structured as some computation kernels that operate on sequences of data records called streams<sup>3</sup>. Imagine achieve high performance when the stream applications<sup>4,5</sup> present the fine locality in LRF and SRF to fully utilize so many ALUs. However, the straightforward coding of scientific programs does not exhibit sufficient locality to effectively exploit the tremendous processing power of Imagine. Therefore, in this paper, we explore a novel locality optimizing algorithm for developing stream programs in Imagine to

achieve high memory performance. Our specific contributions include that we formulate the relationship between streams and kernels as the Data&Computation Matrix (D&C Matrix), and present the key techniques for locality enhancement based on this matrix. We implement our algorithm to five representative scientific applications on the ISIM simulation of Imagine. The results show that our algorithm can effectively improve the computational intensiveness and avoid the utilization of index streams to achieve high locality in Imagine.

## 2. D&C MATRIX

Loops and arrays are fundamental structures of most scientific applications. Thus our approach is based on building a matrix called the Data&Computation Matrix (D&C Matrix) for a given program shown in Fig. 1. Each row of the D&C Matrix represents an array and each column of this matrix describes the access pattern of a loop. The item in the  $i^{\text{th}}$  row denoted  $D_i$  and the  $j^{\text{th}}$  column denoted  $L_j$  position of the D&C Matrix corresponds to a mapping denoted as  $m_{ij}: D_i \rightarrow L_j$ . We define the computation distance  $Cdistance(x,y)$  as the number of iterations between  $x$  and  $y$  such that  $Cdistance(x,y)=y-x$  and the data distance  $Ddistance(c,d)$  as the interval between the two data layouts such that  $Ddistance(c,d)=d-c$ .

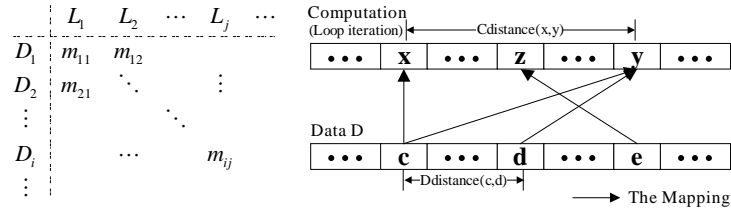


Figure 1. The D&C Matrix and the mapping in the matrix

Furthermore, we treat loop iteration spaces unrolling as the stream organization pattern. We formulate this approach as follows, where  $ORG(i,j)$  denotes the stream organization, the symbol “ $\rightarrow$ ” denotes the connection of different data sequences,  $\max(x)$  is the maximum iteration of the loop body.

$$ORG(i,j) = \sum_{x=0}^{\max(x)} m_{ij}^{-1}(x \mid x \in L_j)$$

Thus, the layout of the basic stream<sup>6</sup> is important for it affects the stream organization. By analyzing the D&C Matrix, form the basic streams according to the least common array region of the most time-consuming loops. We formulate the basic stream layout of each array as follows, where

$f$  is the time-consuming factor that presents the importance of each loop for basic stream layout.

$$BS(i) = \forall j \left( \bigcap \left( \text{ORG}(i, j) \cdot \frac{1}{f} \right) \right)$$

### 3. PROGRAMMING OPTIMIZATION FOR LOCALITY ENHANCEMENT

#### 3.1 Improving LRF Locality

The enhancement of LRF locality can reduce wire delay between clusters, improve computational intensiveness and increase the utilization of LRF<sup>7</sup>. Thus we present the locality optimizations of LRF for shortening the computation distances and the data distances in the D&C Matrix.

##### 3.1.1 Enhancing temporal locality in LRF

Enhancing LRF temporal locality can increase the computational intensiveness to sustain high computational ability. We provide the following formula for the fine temporal locality in LRF.

$$\forall i \forall j \left( \forall x \in L_j \left( m_{ij}^{-1}(x) = m_{ij}^{-1}(x \pm 1) \right) \mid D_i \in D(K) \cap L_j \in L(K) \right)$$

First, aiming at producing computational intensive kernels, we centralize all the computations that perform on the same stream into a large kernel based on the following formula, and yield a new computational intensive matrix.

$$\forall j_1 j_2 \in L_j \left( \exists a \in D_i \left( (m_{ij_1}(a) \cap m_{ij_2}(a)) \neq \phi \right) \right)$$

Second, we consider reducing the computation distance in the new D&C Matrix as follows.

##### 1. Avoiding wire delay between clusters

Due to wire delay becoming increasingly important in locality enhancement, we can't assign the dependent data to different clusters but to the same cluster. There is no influence of wire delay when the following formula is satisfied, where  $\text{cluster}_i$  denotes the records in the  $i^{\text{th}}$  cluster.

$$\forall i \forall j \left( m_{ij}^{-1}(m_{ij}(\text{cluster}_x)) \cap \{ \text{cluster}_y \mid y = 0 \dots 7 \ \& \ y \neq x \} = \phi \right)$$

##### 2. Shortening the computation distance between loops

Data dependence tells us that two references point to the same LRF location, thus the computation distance can be shortened by eliminating the loop-carried dependence<sup>8</sup>. If the dependence can't be eliminated, we consider tiling the computation space<sup>9</sup>. Thus the dependences just exist within inner loops. The left part of Fig. 2 shows the optimizations.

### 3. Reducing the dependent threshold in the inner loop

To achieve fine-grained optimization, we need reduce the computation distance in the innermost loop, that is, reduce the dependent threshold. We can eliminate the intermediate variables to centralize the computations on the same record as many as possible.

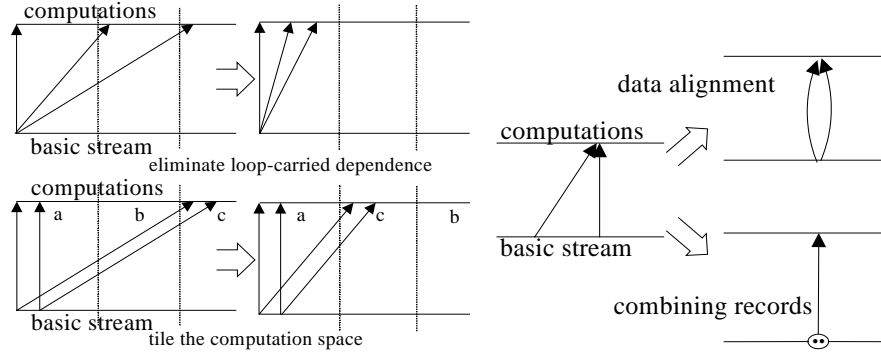


Figure 2. Enhancing temporal and spatial locality in LRF

#### 3.1.2 Enhancing spatial locality in LRF

Since LRF can't make random access, the spatial locality in LRF is successive and limited to the LRF capacity and the overhead caused by SPs<sup>10</sup>. We provide the following formula for fine kernel spatial locality.

$$\forall i \forall j (\forall a \in D_i (m_{ij}(a) = m_{ij}(a \pm 1)) \mid D_i \in D(K) \cap L_j \in L(K))$$

##### 1. Enhancing spatial locality of long stream

We must avoid the very latter part of a long stream reusing the previous data due to the limited LRF capability and SPs. One side, the loop-carried dependence need be eliminated for avoiding the LRF reuse<sup>11</sup>; on the other side, tile the data space and restructure the data stream in LRF so that a single strip length fits in LRF and reduce the utilization of SPs.

##### 2. Shortening the data distance in the inner loop

The iterations of the inner loop need to be placed into a cluster to enhance high spatial locality. First, we can align different records referenced by the same computation or combine these records to a big record<sup>12</sup>, and thereby the data distance can be reduced so that achieve fine spatial locality in LRF, which is shown in the right part of Fig. 2. Second, we can apply loop interchange to reduce the data distance according to the access pattern of the basic stream. Last, we need to reduce the intermediate variables to

utilize the fewest SPs. We formulize the number of SPs kept before iteration  $y$ , where  $\text{NUM}(X)$  denotes the number of sequence  $X$ .

$$\sum \text{NUM}(m_{ij}^{-1}(z)) | \forall i(\forall z > y)(\exists z(m_{ij}^{-1}(z) < m_{ij}^{-1}(y)) \cap (D_i \in D(K)))$$

### 3.2 Improving SRF Locality

The locality in SRF is exposed by forwarding the streams produced by one kernel to subsequent kernels<sup>13</sup>.

#### 3.2.1 Unifying streams between kernels

First, we alter the streams' region to make the streams in successive kernels uniform shown in the left of Fig. 3. Then we can transfer some computations to the next kernel to reduce the intermediate results given in the right of Fig. 3. Last if some parts of a long stream can be reused between kernels, we consider strip-mining the stream to enhance SRF locality.

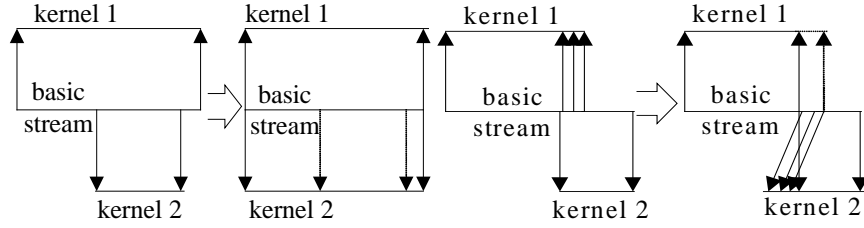


Figure 3. Unifying streams between kernels

#### 3.2.2 Make full use of the SRF capacity

Above all if some loops in the successive kernels exist data dependency, we can transfer the loops in the previous kernel to the next kernel. This idea can reduce the intermediate results to make full use of the SRF capacity and enhance SRF reuse. Second, if some parts of a long stream can be reused between multiple kernels, we consider strip-mining the stream to enhance SRF locality and reduce off-chip memory access overhead.

#### 3.2.3 Avoiding the utilization of index stream

The usage of index streams makes stream organization flexibly, but it also loses the SRF locality owing to too much extra overhead of DRAM reordering. So we must avoid using index stream as follows.

### 1. Organizing streams as the basic stream

To avoid using index stream that reduces SRF locality, we need select successive basic streams as operation objects of kernels by using loop interchange etc. transformations.

### 2. Data-centric loop splitting

We bring forward a new transformation to avoid index stream for higher locality, which is data-centric loop splitting. We can distill the computations that reuse data with large temporal span as self-governed loop.

## 3.3 A locality optimizing algorithm

In this section, we develop a locality optimizing algorithm for stream program generation denoted LOA shown in Fig. 4.

#### ALGORITHM: LOA

**INPUT:** The serial program P

**OUTPUT:** A stream program with fine locality  
form the D&C Matrix of P, denoted M  
ProgramLocality(M)

for each  $L_j$   
KernelLocality( $\forall i(M_{ij})$ )

#### ALGORITHM: ProgramLocality(M)

**INPUT:** The D&C Matrix of P, denoted M

**OUTPUT:** A new D&C Matrix

for each j  
for each c and d  
if  $M_{cj} \circ 0$  &  $M_{dj} \circ 0$   
distribute( $L_j$ )  
for each i  
for each x and y  
if  $M_{ix} \circ 0$  &  $M_{iy} \circ 0$   
merge( $L_x, L_y$ )  
for each j and i  
if  $\text{output}(M_{ij}^{-1}(L_j)) \sim \text{input}(M_{ij+1}^{-1}(L_{j+1})) \circ \#$   
schedule( $D_i, L_{j+1}$ )  
for each j and i  
common =  $M_{ij}^{-1}(L_j) \sim M_{ij+1}^{-1}(L_{j+1})$   
if common  $\circ \#$  then begin  
if common > T  
unify( $M_{ij}^{-1}(L_j), M_{ij+1}^{-1}(L_{j+1})$ )  
else  
striping( $M_{ij}^{-1}(L_j), \text{common}$ )  
striping( $M_{ij+1}^{-1}(L_{j+1}), \text{common}$ )  
for each i and j  
$$BS(i) = \forall j \left( \bigcap \left( \sum_{x=0}^{\max(x)} m_{ij}^{-1}(x \mid x \in L_j) \cdot \frac{1}{f_i} \right) \right)$$

#### ALGORITHM: KernelLocality( $\forall i(M_{ij})$ )

**INPUT:**  $\forall i(M_{ij})$  is the  $j^{\text{th}}$  column of M

**OUTPUT:** the kernel with fine locality

for each i  
if  $\sum_{x=0}^{\max(x)} m_{ij}^{-1}(x \mid x \in L_j) \neq BS_i$   
avoid the utilization of index streams  
for each c,  $d \in D_i$   
if  $M_{ij}(c) = M_{ij}(d)$   
put c and d on the same cluster  
judge if the dependence can be eliminated  
if success then  
eliminate the loop-carried dependence  
InnerLoopLocality( $\forall i(M_{ij})$ )  
else  
tiling the loop  
InnerLoopLocality( $\forall i(M_{ij})$ )

#### ALGORITHM: InnerLoopLocality( $\forall i(M_{ij})$ )

**INPUT:**  $\forall i(M_{ij})$  is the  $j^{\text{th}}$  column of M

**OUTPUT:** the loop with fine locality

for each i  
for each c  $\in D_i$  then  
reduce the intermediate variables  
between two computations on c  
for each d  $\in D_i$  &  $d \circ c$  then begin  
if  $M_{ij}(c) = M_{ij}(d)$   
combine c and d to a big record  
or data alignment  
for each x, y  $\in L_j$   
if  $M_{ij}^{-1}(x) = M_{ij}^{-1}(y)$   
reduce Cdistance(x,y)

Figure 4. The LOA algorithm

First, **LOA** algorithm forms the D&C Matrix of the program that need to be expressed as stream program. Then apply **ProgramLocality** algorithm for data-centric program restructuring to enhance the locality and computational intensiveness between kernels. Afterward it employs the routine **KernelLocality** to optimize the locality of each kernel.

**ProgramLocality** first analyze the profile information to find the most time-consuming parts that need be streaming. And increase the computations per words by loop distribution and loop fusion based on the D&C Matrix transformations. Then schedule partial computations to the next kernel and reuse the same data region between kernels by unifying stream or strip-mining stream to enhance the SRF locality. Last, form the basic streams according to least common array region of high access frequency by profiling.

**KernelLocality** first avoids the utilization of index streams to enhance the SRF locality. Then optimize the LRF locality. It assigns the dependent data to the same cluster to avoid the wire delay between kernels. And increase the locality of long streams by eliminating the loop-carried dependence between loops or tiling the computation space. Finally, invoke **InnerLoopLocality** to reduce the computation distance and data distance in the inner loop by eliminating the intermediate variables, enlarging the records, aligning different records and reducing the dependent threshold.

#### 4. EXPERIMENTAL RESULTS AND ANALYSIS

Five representative scientific programs are used to evaluate our LOA algorithm on ISIM simulator<sup>14</sup> that is a cycle-accurate simulator of Imagine, including Swim, Dfft, Transp, Vpenta and N-S. Swim is a weather prediction program for comparing the performance of current supercomputers in SPEC2000. Dfft and Transp are the most time-consuming subroutines in Capao that is an application on the field of optics. Vpenta is one of the kernels in nasa7. N-S is an application of solving partial differential equation of fluid dynamics for the flow of an incompressible viscous fluid.

Fig. 5 shows the computational intensiveness that is a significant representation of LRF locality by applying our LOA algorithm. Groups of bars represent the original version (Orig) of each application and the version optimized with LOA algorithm (LOA). We can observe the LOA optimization improves the computations per memory accesses of the five programs. However Swim and Transp achieve a little varying, because Swim has too many data and irregular access pattern so that the loops are difficult to be distributed or combined, and all the arrays in Transp are referenced rarely leading a little variety of computational intensiveness compared with

original stream program. The LOA optimization can centralize all computations in Vpenta, Dfft and N-S to a kernel due to repetitive references to each array in these programs.

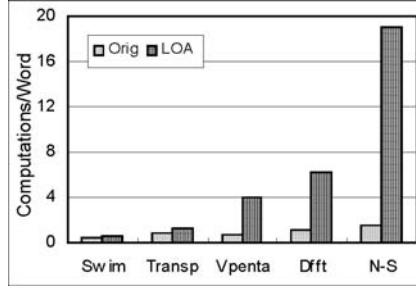


Figure 5. Computational intensiveness

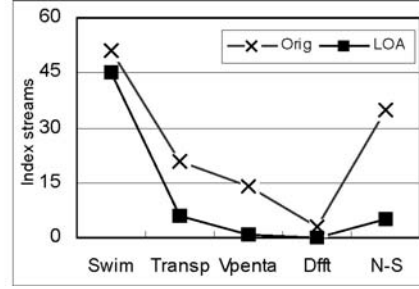


Figure 6. The reduction of index streams

Fig. 6 shows the reduction of index streams by applying LOA to present the variety of SRF locality. One of the key techniques in LOA is to form appropriate basic streams so that the number of index streams can be reduced. But in Swim, the choice of basic stream has little effect on stream forming owing to complex data access pattern. Dfft has a few data in original stream program, so the index streams are lessened a little too. The index streams of Transp, Vpenta and N-S reduce observably for achieving higher performance. In Transp, lessening the scale of original basic streams at the beginning of this program can avoid a great deal of index streams. The index stream can be eliminated in Vpenta by applying MBO when stream is short due to regular data access pattern by using SPs in kernel. In N-S, the basic stream reorganization plays an important role of reducing index streams.

Fig. 7 presents the variety of computation rate of these applications measured in the number of operations executed per second by applying LOA optimization. Our LOA optimization assigns all dependent data to a cluster, avoiding communication delay and memory access latency. However Swim optimized by LOA still presents overfull index streams so that memory delay can't be overlapped, resulting in low computation rate. Despite Transp and Vpenta both achieve higher LRF locality by eliminating loop-carried dependence between inner loops and shortening dependent threshold in inner loop, their computation rate are increased a little, because both low computational intensiveness of Transp and the usage of index streams in Vpenta when streams are long make overlapping memory latency difficultly. N-S also presents high computational density by applying LOA optimization, however the computation rate of N-S is slow because it invokes inefficient



mathematical kernels for many times. The high computation rate of Dfft indicates that the stream programming system delivers high computational density on this application.

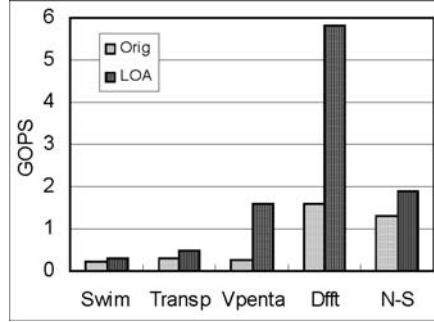


Figure 7. The variety of computation rate

Table 1 illustrates the efficiency of the program yielded by our optimization (LOA) compared with original stream program (Orig). It is obvious that our optimization provides high speedup of Dfft, Transp, Vpenta and N-S due to fine locality. And compared with highly sensitive to memory latency of general processor, these applications can hide latency to achieve good performance. But for data intensive applications such as Swim, the speedup is low due to irregular access pattern so that our optimization can't hide memory access latency. In conclusion, Swim is not well suited for the Imagine architecture.

Table 1. Comparison of different scientific applications by applying LOA

	Swim	Transp	Vpenta	Dfft	N-S
Cycles(Orig)	8.10E+09	1.98E+07	4.97E+07	5.07E+11	1.68E+08
Cycles(LOA)	6.69E+09	9.28E+06	1.69E+07	9.71E+10	4.36E+08
Speedup	1.21	2.13	2.94	5.22	2.60

## 5. CONCLUSION AND FUTURE WORK

In this paper, we explore a novel locality optimizing algorithm for developing stream programs in Imagine to fully sustain high computational ability. Our specific contributions include that we formulate the relationship between streams and kernels as the Data&Computation Matrix (D&C Matrix), and present the key techniques for locality enhancement based on

this matrix. The results show that our algorithm can achieve high locality in LRF and SRF.

One future work is to research more programming optimizations to exploit more architectural features of Imagine. Another is to search more scientific applications suited for stream architecture by applying our algorithm.

## ACKNOWLEDGEMENTS.

We gratefully thank the Stanford Imagine team for the use of their compilers and simulators and their generous help. We also acknowledge the reviewers for their insightful comments.

## REFERENCES

1. Saman Amarasinghe, William. Stream Architectures. In PACT03, September 27, 2003.
2. Bruce Khailany. The VLSI Implementation and Evaluation of Area-and Energy-Efficient Streaming Media Processors. Ph.D. thesis, Stanford University, 2003.
3. Ola Johnsson, et al. Programming & Implementation of Streaming Applications. Master's thesis, Computer and Electrical Engineering Halmstad University, 2005.
4. B. Khailany et al. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, March 2001.
5. Saman Amarasinghe et al. Stream Languages and Programming Models. In PACT03, September 27, 2003.
6. Peter Raymond Mattson. A Programming System for the Imagine Media Processor. Dept. of Electrical Engineering. Ph.D. thesis, Stanford University, 2002.
7. Nuwan S. Jayasena. Memory Hierarchy Design for Stream Computing. Ph.D. thesis, Stanford University, 2005.
8. M. J. Wolfe. High Performance Compilers for Parallel Computing. Addison-Wesley, 1996.
9. J. Xue. Loop Tiling for Parallelism. Kluwer Academic Publishers, Boston, 2000.
10. Jinwoo Suh, Eun-Gyu Kim, Stephen P. Crago, Lakshmi Srinivasan, and Matthew C. French. A Performance Analysis of PIM, Stream Processing, and Tiled Processing on Memory-Intensive Signal Processing Kernels. In ISCA03, 2003.
11. M. E. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452 – 471, October 1991.
12. Jing Du, Xuejun Yang, et al. Scientific Computing Applications on the Imagine Stream Processor. In ACSAC06, September 6-8, 2006.
13. Jung Ho Ahn, William J. Dally, et al. Evaluating the Imagine Stream Architecture. In ISCA04, 2004.
14. Abhishek Das, Peter Mattson, et al. Imagine Programming System User's Guide 2.0. June 2004.

Distributed and Parallel Systems

From Cluster to Grid Computing

Kacsuk, P.; Fahringer, Th.; Nemeth, Z. (Eds.)

2007, XII, 223 p. 64 illus., Hardcover

ISBN: 978-0-387-69857-1